1. a) We essentially need to calculate a weighted average using the percentages in table.

$$CPI = 1.2\,\tfrac{cc}{instr}(0.15) + 1.1\,\tfrac{cc}{instr.}(0.45) + 1.3\,\tfrac{cc}{instr.}(0.40) = \underline{1.195\,\tfrac{cc}{instr.}}$$

for control instructions → 10% + 1% + 2% + 2% = 15% = 0.15

for alu instructions → 22% + 2% + 6% + 9% + 6% = 45% = 0.45

for loads & stores → 21% + 9% + 8% + 2% = 40% = 0.40

b) We can use units to help remember the proper equation here. We want execution time in seconds and we have # of instructions, cc/instr., and cc/s

$$\text{execution time} = \frac{1}{\cancel{cc}/s} \cdot \frac{\cancel{cc}}{instr.} \cdot instr.$$

$$= \frac{1.195\ cc/instr. \times 2 \cdot 10^{6}\ instr.}{2.5 \cdot 10^{9}\ cc/s} = 0.956 \cdot 10^{-3}\ s$$

$$= \underline{0.956\ ms}$$

c) We basically recalculate a) & b) w/ new numbers

$$CPI = 1.2\ cc/instr\,(0.40) + 1.1\ cc/instr.\,(0.15) + 0.9\ cc/inst\,(0.45)$$

$$= 1.05\ cc/instr$$

$$\text{execution time} = \frac{1.05\ cc/instr. \times 2 \cdot 10^{6}\ instr.}{3.5 \cdot 10^{9}\ cc/s} = \underline{0.6\ ms}$$

$$\text{speedup} = \frac{0.956\ ms}{0.6\ ms} = \underline{1.59\times}$$

2. We can use Figure 2.14 to translate register names to their associated register numbers and Figure 2.19 to determine opcodes and function fields. I realize the problem does not ask for function fields, but I will provide them anyway.

add $t4, $t5, $t6

rd = $t4 = 12_{10} opcode = R-type = 0
rs = $t5 = 13_{10}
rt = $t6 = 14_{10} function field = 100000_2
= 32_{10}

the 10 indicates base 10 number

lw $t2, 8($t4)

rs = $t4 = 12_{10} opcode = 100011_2
rt = $t2 = 10_{10} = 35_{10}
immediate = 8_{10}

lw $t3, 12($t4)

rs = = $t4 = 12_{10} opcode = 35_{10}
rt = $t3 = 11_{10}
immediate = 12_{10}

xor $t6, $t2, $t3

rd = $t6 = 14_{10} opcode = 0
rs = $t2 = 10_{10} function = 100110_2
rt = $t3 = 11_{10} = 38_{10}

addi $t4, $t4, 8

rt = $t4 = 12_{10} immediate = 8_{10}
rs = $t4 = 12_{10} opcode = 001000_2
= 8_{10}

2. continued...

sw \$t6, 24 (\$t4)

$rt = \$t6 = 14_{10}$ $\quad$ $opcode = 101011_2$

$rs = \$t4 = 12_{10}$ $\quad$ $= 43_{10}$

$immediate = 24_{10}$

3.

$$\underbrace{000000}_{\text{R-type}}\;\underbrace{10000}_{rs = 16_{10}}\;\underbrace{10000}_{rt = 16_{10}}\;\underbrace{10000}_{rd = 16_{10}}\;\underbrace{00000}_{\substack{shamt\\=0}}\;\underbrace{100000}_{\substack{function\ 32_{10}\\add}}$$

R-type add \$s0, \$s0, \$s0

4. this is an I-type of instruction w/ an op code of 101011_2

sw \$t0, -16(\$sp)

$rs = 29_{10} = \$sp = 11101_2$

$rt = \$t0 = 8_{10} = 01000_2$

immediate = -16

$= 1111111111110000_2$

$16_{10} = 000010000_2$

to get -16_{10} we can determine 16_{10} in base 2 and then 2s complement the result, which means invert the bits & add 1

$$\begin{array}{r} -16_{10} = 111101111 \\ + \quad\quad\quad 1 \\ \hline 111110000 \end{array}$$

$$\underbrace{101011}_{op}\;\underbrace{11101}_{rs}\;\underbrace{01000}_{rt}\;\underbrace{111111111111}_{\substack{immediate\\15:4}}$$

$$\underbrace{0000}_{immediate\ 3:0}$$

5. a) the address in hex is: 0x20014924

the psudo instruction to do this would be:

```
la $t1, 0x20014924
```

this turns into the following:

```
lui $t0, 0x2001
ori $t0, $t0, 0x4924
```

b) no you cannot because the jump format only allows the encoding of a 26-bit pseudo immediate value that is shifted to the left by 2 bits giving us 28 bits. Thus, the remaining 4 bits must come from PC+4, PC+4 in this case is 0x00000004 and thus the upper 4 bits are 0000_2 which is not equal to 0010_2

c) same as b) so no

d) PC+4 = 0x1FFFF004

upper 4 bits are $0001_2 \neq 0010_2$

so no

e) you would use the following assembly code:

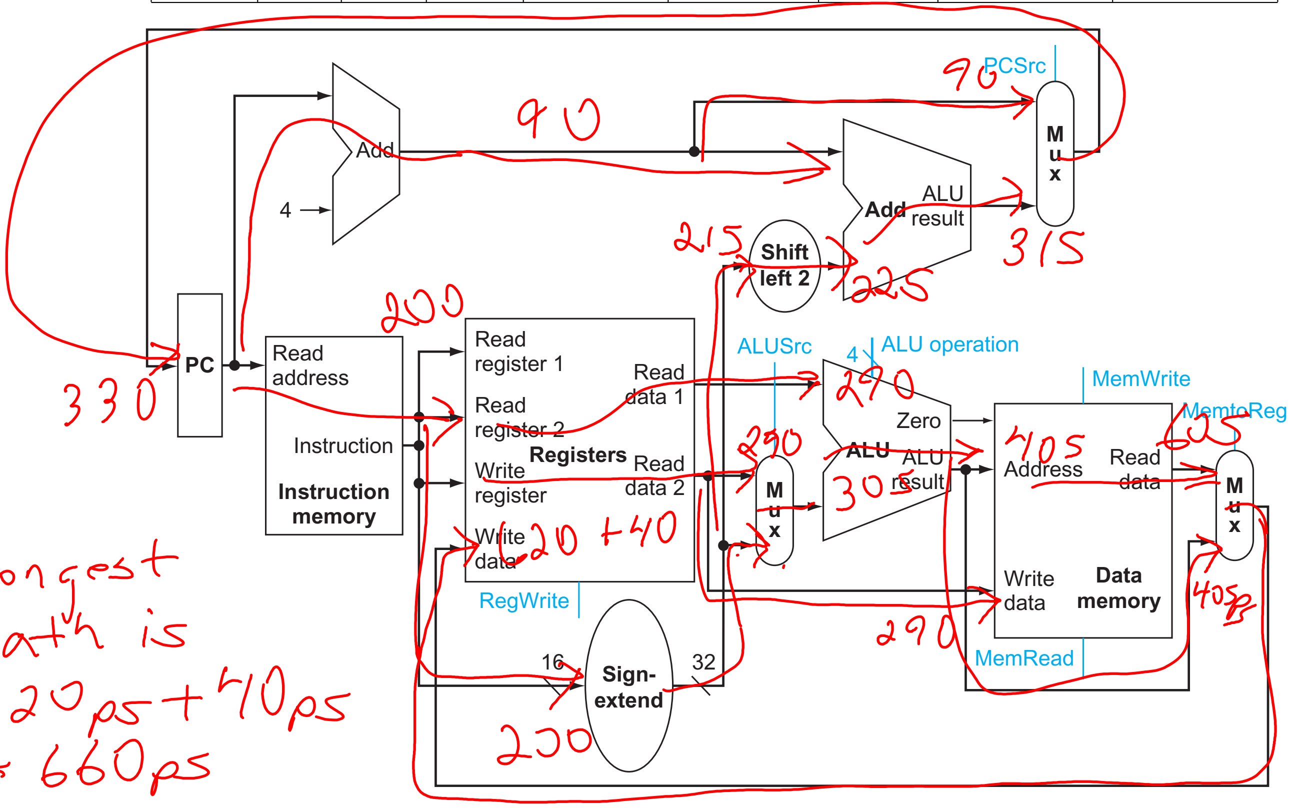
```
la $at, 0x20014924
jr $at
```

or

```
lui $at, 0x2001
ori $at, $at, 0x4924
jr $at
```

6. **(20 points)** Consider the single-cycle datapath shown below with the following component latencies:

I-mem	Add	Mux	ALU	Reg Read	Reg Write	D-Mem	Sign-Extend	Shift-Left-2
200ps	90ps	15ps	100ps	90ps	40ps	200ps	15ps	10ps



longest path is 620ps + 40ps = 660ps

(a) What is the maximum achievable clock rate for the datapath above? Ignore the effects of the control unit.
Hint: Mark the arrival time of valid data on each of the components in the figure above.

f = 1/660ps = 1.51 GHz

(b) What is the latency of a load instruction in this datapath? 660 ps

(c) What is the latency of a store instruction? 405ps + 200ps = 605ps

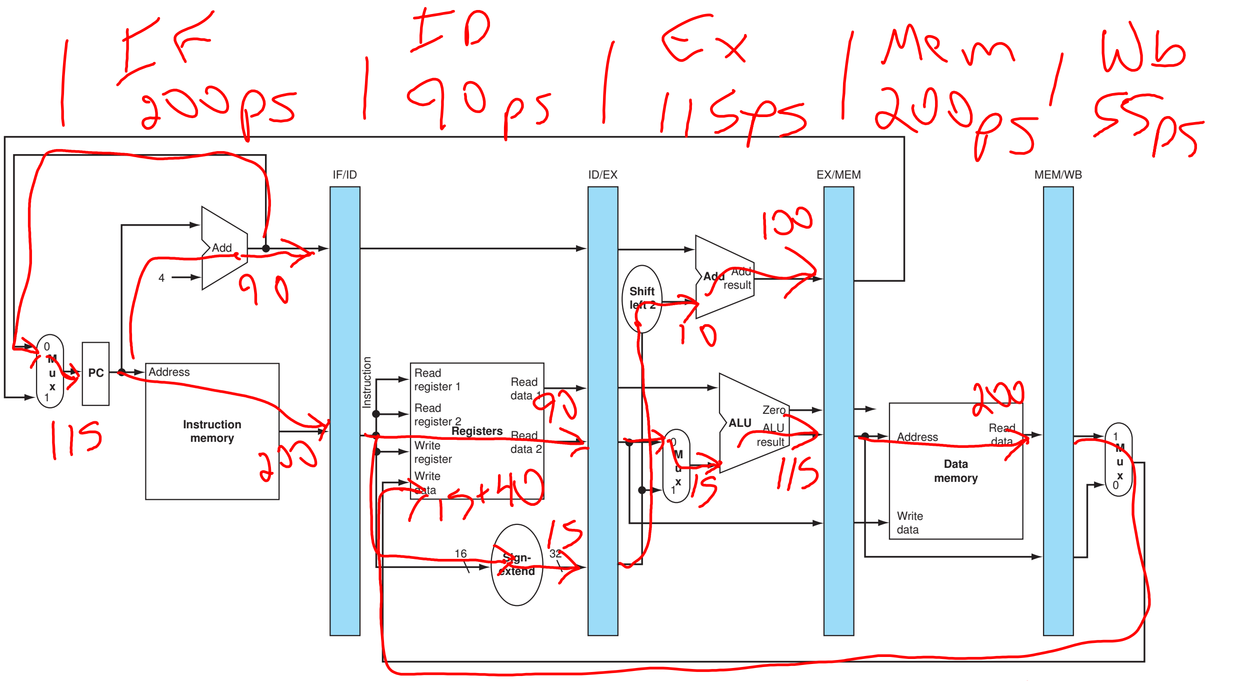
(d) What is the latency of an ALU instruction? 405ps + 15ps + 40ps = 460ps

(e) What is the CPI of this design? 1 cc/instr

(f) Consider the pipelined datapath shown below. Ignoring the effects of the control unit and flip-flop delays, what is the maximum achievable clock rate?

The critical path is 200ps so the clock rate would be

$\frac{1}{200ps}$ = 5 GHz



(g) What would be the latency of a load instruction in the pipelined datapath? 5×200ps = 1000ps

(h) What would be the latency of an ALU instruction? the same 1000ps

(i) Assuming a long running program with no pipeline hazards, what is the instruction throughput of the above pipelined processor? What is the speedup compared to the single-cycle machine?

(j) What is the maximum achievable clock rate of the pipelined datapath above if we account for a 20ps flip-flop delay.

(k) Why is the single-cycle design impractical?

(l) Compare the speedup you calculated in Problem 6i with that of the ideal speedup for a 5-stage pipeline. What are some things that reduce the speedup of pipelining?

i) instruction throughput idealy would be 1instr/cc or 1cc/instr. The speed up would then be

$\frac{5}{1.515} = \underline{3.3x}$

j) the critical path would then be

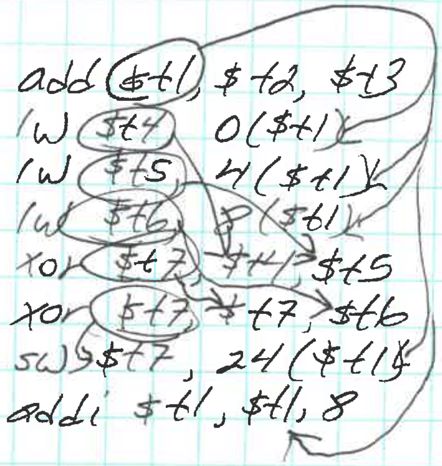
200ps + 20ps = 220ps

clock rate = 1/220ps = $\underline{4.55\text{GHz}}$

k) The critical path is too long w/ a single cycle and many components remain idle throughout most of the cycle. not efficient and slower clock.

l) In part i) we calculated a 3.3x speedup whereas ideal is 5x. The main loss here was pipeline fragmentation. Wb only required 55ps whereas Mem took 200ps. the stages are not broken up evenly. other inefficiencies can be due to hazards and flip-flop delays, clock skew, setup time etc.

7,



```
add $t1, $t2, $t3
lw $t4, 0($t1)
lw $t5, 4($t1)
lw $t6, 8($t1)
xor $t7, $t4, $t5
xor $t7, $t7, $t6
sw $t7, 24($t1)
addi $t1, $t1, 8
```

a) see above. for example:

```
add $t1, $t2, $t3
lw $t4, 0($t1)
```

means that add produces $t1 and lw consumes $t1

b. Assuming register write happens at the beginning of the clock cycle and read happens at the end means that if an instruction is producing a register needed by another instruction and is in the WB stage when the other instruction is in the decode, no stalling is needed and data is passed through the register file as ~~expected~~ expected.

```
add $t1, $t2, $t3
nop
nop
lw $t4, 0($t1)
lw $t5, 4($t1)
lw $t6, 8($t1) ↙
nop → xor $t7, $t4, $t5
nop
nop
xor $t7, $t7, $t6
nop
nop
sw $t7, 24($t1)
```

I forgot there should be a nop here because of $t5 between lw and xor

7. continued...

c.

```
add $t1, $t2, $t3
lw $t4, 0($t1)
lw $t5, 4($t1)
lw $t6, 8($t1)
xor $t7, $t4, $t5
xor $t7, $t7, $t6
sw $t7, 24($t1)
addi $t1, $t1, 8
```

it turns out we don't need nops if we have data forward with this sequence of code.

The ~~third~~ second lw produces $t5 and the 1st xor consumes $t5, but the third lw seperates the two so we don't need a nop!

Similarly the last lw produces $t6 and the second xor consumes $t6, but the first xor seperates the two so no need for a nop. All other dependences can be forwarded w/ out ~~no~~ nops.

d. same as above because no nops.

8. a)

cc	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
lw $t2, 0($t1)	F	D	E	M	W											
beq $t2, $t0, label2		F	D	D	E	M	W									
lw $t3, 0($t2)			F	F	D	E	M	W								
beq $t3, $t0, label1					F	D	D	E	M	W						
add $t1, $t3, $t1						F	F	D	E	M	W					
beq $t2, $t0, lable2								X	F	D	E	M	W			
lw $t3, 0($t2)										F	D	E	M	W		
sw $t1, 0($t2)											X	F	D	E	M	W

flushes due to branches being taken

b) This code requires 16 clock cycles to execute to completion.

cc	1	2	3	4	5	6	7	8	9	10	11	12	13	14
lw $t2, 0($t1)	F	D	E	M	W									
beq $t2, $t0, label2		F	D	D	E	M	W							
lw $t3, 0($t2)			F	F	D	E	M	W						
beq $t3, $t0, label1					F	D	D	E	M	W				
add $t1, $t3, $t1						F	F	D	E	M	W			
beq $t2, $t0, label2								F	D	E	M	W		
lw $t3, 0($t2)									F	D	E	M	W	
sw $t1, 0($t2)										F	D	E	M	W

no flush because 100% correct prediction!

8. continued...

d) the 100% predictor allows the code to execute in 14cc as opposed to 16cc so speedup is:

$$\frac{16}{14} = \underline{1.1429x}$$

e) If you move the ~~decision~~ resolution ~~[illegible]~~ (decision) to the decode stage, you actually don't need the predictor but you would have additional forwarding and stalling

```
                                              9 10 11 12 13 14 15 16
                cc  1 2 3 4 5 6 7 8
lw $t2, 0($t1)      F D E (M) W            2 stalls are needed because
beq $t2, $t0, label2  F D D D E M W          $t2 is now needed in decode!
lw $t3, 0($t2)          F F F D E (M) W          same here because of $t3
beq $t3, $t0, label1          F D ● ● E M W
add $t6, $t3, $t1               F F F D E M W
beq $t2, $t0, label2                  F D E M W
lw $t3, 0($t2)        no ~~stall~~ flushes   F D E M W
sw $t1, 0($t2)        because branch          F D E M W
                      is resolved in decode!
```

this scheme takes 16 cc to execute, which is the same as part a). Although we avoid two flush cycles ~~[illegible]~~ by resolving branches earlier in the pipeline, we introduce 2 additional stalls due to making data hazards worse. Best solution so far is to use a branch predictor!

f) We can use the tables in Figures 2.14 & 2.19 to determine the opcode & rs, rt encoding.

opcode for beq: 000100_2

beq $t2, $t0, label2 $rs = \$t2 = 10_{10} = 01010_2$

$rt = \$t0 = 8_{10} = 01000_2$

for the offset, label 2 is 3 instructions ahead of the branch delay so the offset is 3

8. f) continued...

beq $t2, $t0, label2

$\underbrace{0001\,00}_{\text{opcode}}\underbrace{00\,010}_{\text{rs}}\underbrace{0\,1000}_{\text{rt}}\underbrace{0000\,0000\,0000\,0011}_{\text{offset}}$

in hex!

$\underline{0x11480003}$

beq $t3, $t0, label1

$rs = \$t3 = 11_{10} = 01011_2$
$rt = \$t0 = 8_{10} = 01000_2$

the offset is -3 because label1 is 3 instructions behind the branch delay, i.e. PC+4

$\underbrace{0001\,00}_{\text{opcode}}\underbrace{00\,101}_{\text{rs}}\underbrace{1\,0100}_{\text{rt}}\underbrace{0\,1111\,1111\,1111\,101}_{\text{offset}}$

$\underline{0x1168FFFD}$

9. a) The delays without the multiplier are the same as those in problem 6 so the cycle time without the multiplier is 660ps & the clock rate is 1.51GHz

With the multiplier, the ALU delay goes from 100ps to 300ps. Thus, the critical path goes up 200ps and is $\underline{860ps}$ with the multiplier
The clock rate is $\underline{1.163\,GHz}$ which is $\frac{1.51}{1.163} = \underline{1.30x}$ slower!

b) The clock rate is 1.30x slower but the instruction count was reduced by 20%. Also note the CPI remains 1.

$$\frac{t_{new}}{t_{old}} = 1.3 \times 0.8 = 1.04$$ or 4% slower

$$\text{speedup} = \frac{t_{old}}{t_{new}} = \frac{1}{1.04} = \underline{0.962}$$

9. continued...

c) If our ALU now has a 300ps delay, the critical path of the processor is now through the execute stage w/ a delay of 315ps.

The clock rate of the pipelined processor is now

$\frac{1}{315ps} = \underline{3.175 GHz}$

d) The clock rate is now $\frac{4.55}{3.175} = 1.43x$ slower!

Overall speedup = $\frac{1}{0.8 \times 1.43} = 0.872x$ speedup

so no! the multiplier slows the design down.

e) By pipelining the multiplier and pulling it out of the ALU, it is no longer on the critical path so the clock rate remains unchanged with the addition of the multiplier so $\underline{4.55 GHz}$

f) The problem basically asks use to assume the CPI does <u>not</u> change by pipelining the multiplier. If that is the case, we will run 20% faster so our speed up is $\frac{1}{0.8} = \underline{1.25x}$